

RL-TR-95-295, Vol I (of four)
Final Technical Report
April 1996



ROMULUS, A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT: ROMULUS OVERVIEW

Odyssey Research Associates, Inc.

S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird,
D. Long, D. McCullough, I. Meisels, D. Rosenthal,
I. Sutherland, and A. Weitzman

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19960724 061

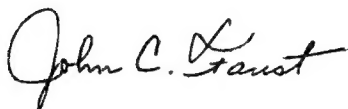
DTIC QUALITY INSPECTED 3

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR- 95-295, Vol I (of four), has been reviewed and is approved for publication.

APPROVED:



JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3AB), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE April 1996		3. REPORT TYPE AND DATES COVERED Final Aug 90 - Jun 94	
4. TITLE AND SUBTITLE ROMULUS, A COMPUTER SECURITY PROPERTIES MODEL ENVIRONMENT: Romulus Overview				5. FUNDING NUMBERS C - F30602-90-C-0092 PE - 35167G PR - 1065 TA - 01 WU - 03	
6. AUTHOR(S) S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird, D. Long, D. McCullough, I. Meisels, D. Rosenthal, I. Sutherland, and A. Weitzman					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates, Inc. 301 Dates Drive Ithaca NY 14850-1326				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Rd Rome NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-95-295, Vol I (of four)	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: John C. Faust/C3AB/(315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Romulus security properties modeling environment contains tools, theories, and models that support the high-level design and analysis of secure systems. The Romulus nondisclosure tool supports development and analysis of distributed composite security models and their properties. The Romulus modeling approach establishes the models on a solid theoretical basis and uses formal mathematical tools to aid in the analysis. Romulus allows a user to express a model of a secure system using a formal specification notation that combines graphics and text. Verification of the model proves that it satisfies its critical properties. The user verifies the model by using a combination of automatic decision procedures and interactive theorem proving. The primary emphasis in the current system is the analysis of multilevel trusted system models to see if they satisfy nondisclosure properties. Romulus also includes a tool for formally specifying and verifying authentication protocols. This tool can be used to reason about the beliefs of the parties engaged in a protocol in order to analyze whether the protocol achieves the desired behavior. The Romulus theories include formal theories of nondisclosure, integrity, and (see reverse)					
14. SUBJECT TERMS Computer security, Nondisclosure, Integrity, Availability, Security properties modeling, Information flow analysis, Design verification, Authentication protocol analysis, (see reverse)				15. NUMBER OF PAGES 48	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UL	

13. (Cont'd)

availability security. The Romulus library of models demonstrates the application of these theories.

14. (Cont'd)

Multilevel security, Security policy

Preface

This four volume report describes Romulus, a security modeling environment. Romulus includes a tool for constructing graphical hierarchical process representations; an information flow analyzer; a process specification language; and techniques to aid in doing proofs of security properties. Romulus also contains tools for the specification and analysis of authentication protocols. Using Romulus, a user can develop and analyze security models and properties. The foundations of Romulus are formal theories of security; applications of these theories are demonstrated in a library of models.

In describing Romulus in this volume, we assume that the reader is generally familiar with computer security concerns, but not with Romulus, the HOL environment, or the underlying theories of Romulus.

Organization of the Romulus Documentation Set

This volume is Volume I of a four volume documentation set; this volume contains an overview of the Romulus environment. Volume II describes the Romulus theories for nondisclosure, integrity, and availability, and Volume III describes the Romulus library of models. Volume IV is the Romulus User's Manual; it contains descriptions of the Romulus tools, how to use them, and tutorial examples.

Organization of This Volume

This volume presents an overview of Romulus. Chapter 1 briefly describes the security methodology of Romulus and outlines the tools in the Romulus environment. In Chapter 2, we discuss the security properties that Romulus modeling supports. In Chapter 3, we demonstrate the Romulus toolset with some simple examples. In Chapter 4, we summarize the Romulus library of models.

Conventions

This document set uses the following conventions. Computer code, specifications, program names, file names, and similar material are typeset using a typewriter font. Interactive computer sessions are surrounded by a rounded

box. Within this box, user input is typeset using an *italic typewriter* font; computer output is typeset using the `typewriter` font. Some computer output has been reformatted for presentation purposes; it may not appear in this document exactly as it appears on your screen.

Acknowledgements

Portions of this volume were extracted from previous ORA reports and course materials, specifically, [27], [33], and [10].

Contents

1	Introduction	1
1.1	The Romulus Security Methodology	2
2	Security	5
2.1	Nondisclosure	6
2.2	Integrity	8
2.2.1	Integrity Models	9
2.2.2	Authentication Protocols	11
2.3	Availability	11
3	The Romulus Toolset	13
3.1	The Graphical Interface	13
3.1.1	Design	13
3.1.2	Flow Analysis	15
3.2	Proving Security	17
3.3	Authentication Protocol Analysis Tool	19
3.3.1	Describing and Specifying a Protocol	20
3.3.2	Proving a Protocol	21
4	Library of Models	23
4.1	Abstract Guard	23
4.2	MINIX	24
4.3	Network Driver	25
4.4	Authentication Protocols	26
4.5	Distributed Database	27
4.6	Fault Tolerant Reference Monitor	27
4.7	Real-Time Scheduler	28

Bibliography

30

List of Figures

3.1	The simple example	14
3.2	Flow analysis of the simple example	16
3.3	The filter process specification	18

Chapter 1

Introduction

The Romulus Computer Security Properties Modeling Environment is a set of tools for modeling, analyzing, and verifying systems with critical requirements for security. The Romulus models, methodologies, and analysis techniques are based on formal theories of three types of security policies: nondisclosure of information, integrity, and availability (also known as service assurance or avoidance of denial of service). The Romulus system includes a graphical user interface for describing the structure of system models as a collection of communicating components and a specification language for describing the functionality of these components. Additional support for describing and analyzing models is provided by a theorem prover for proving the correctness of components, a library of examples, and a tool for analyzing the information flow properties of the system. The Romulus system also includes tools for the specification and analysis of authentication protocols.

Where possible, the methods and models that Romulus uses are put on a solid mathematical basis. The theoretical bases of nondisclosure security in Romulus are formal theories based on *restrictiveness* [18]. These theories provide a general way of representing systems mathematically as state machines and provide a generic definition of security. The theoretical basis of integrity security in Romulus includes various models of integrity, including a belief logic that is used to describe and verify authentication protocols. The theoretical basis of availability security in Romulus includes formal theories of fault tolerance and timeliness requirements.

1.1 The Romulus Security Methodology

The Romulus security methodology is designed to assist in the modeling of secure systems. The desired properties of these systems should be spelled out in a *security policy*, defined in [8] as a “statement of intent with regard to control over access to and dissemination of information”. How a security policy is put in place is described in a *security model*, which “precisely describes important aspects of security and their relationships to system behavior” [20]. A model serves to provide a clear understanding of those aspects of a system that affect security. In Romulus, we use an additional term, *security theory*, which is a mathematical definition of a security property.

The Romulus environment provides support for a variety of methodologies for investigating different kinds of security properties. Security analysis is not easy, as there are many ways in which security problems can arise. Because of the complexity of security analysis, it is often desirable for the secure system designer to first examine an abstract representation of a system. Such an analysis can shed early insight into potential system design issues. By applying this analysis early in the development process, secure system designers may be able to more rapidly identify potential security problems and thus reduce the overall system development cost. Designers may also be able to surface potential security flaws that would be difficult to detect by manual analysis.

Romulus provides support for secure system analysis in three key ways. First, Romulus supplies the basis for a system security model that describes high-level system security requirements. In this manner, Romulus allows the designer to obtain a clear understanding of the fundamental system security model without being prematurely distracted by implementation detail. The modeling provided by Romulus is particularly relevant to trusted applications that are intended to run on an underlying trusted computing base (TCB). Second, Romulus provides tool support to allow the secure system designer to check that the system model meets important security constraints. Although these checks are not an exhaustive set of requirements to guarantee the security of the implementation, the checks do provide insight into key security problems early in the system development. Finally, the system security model and analysis provide the basis for the security argument of the overall system architecture, which is then supported by further refinement in the detailed design and implementation.

It is important to remember that Romulus contains security modeling tools that are used to model and analyze selected aspects of security. Romulus does not currently provide facilities for mapping the model to code, nor is it a detailed design or code analysis tool.

Romulus is a security modeling environment; it provides support for the development of security models and provides support for the analysis and verification of some of the security properties of these models. This support includes a language for the formal specification of a system model, a means for making formal statements about the desired security properties of the system, and techniques designed to aid in proofs of these properties. The HOL90 [37, 42, 43, 44] proof assistant system is used in the current Romulus release for this purpose. For example, Romulus contains tools for the specification and the proof of nondisclosure properties of processes. Another Romulus tool, the graphical interface, is designed to aid in this process. Romulus also contains tools for the specification and analysis of authentication protocols.

The Romulus nondisclosure security methodology includes definitions of security for system models together with collections of theorems that aid in constructing proofs of security for particular system models. Each definition of nondisclosure security used in Romulus is *composable*. This composability property allows the conditions for inferring the security of a whole system to be expressed as security conditions on its components. We believe that the security enforcement and composability provided by restrictiveness make it an attractive choice for a security policy on trusted systems and processes.

Romulus can be used to analyze the security of systems that handle information at multiple security levels. On a multilevel system, users should not be able to learn through their interaction with the system any sensitive information that they are not authorized to learn. A property that formalizes this condition is *restrictiveness*. Romulus provides special support for showing that systems meet this property. Part of the support is provided by the graphical user interface; another part of this support comes in the form of formal theories and techniques that aid in constructing security proofs.

The Romulus integrity methodology includes various models of integrity. One of these, a belief logic, is the basis for a tool for specifying and analyzing authentication protocols. Authentication protocols are important contributors to integrity assurance because they are used to establish the correct identity of processes and to distribute encryption keys. An authentication protocol is an exchange of messages between a number of processes, called

“principals”. A typical aim of a protocol is to establish an encryption key shared by two principals.

The authentication protocol tool includes a protocol description language, which describes a protocol as a sequence of messages, and a protocol specification language, which specifies not only the messages, but their meanings. This specification language allows one to reason about a protocol to determine if it satisfies its design goals. The tool contains support for constructing formal proofs that the design goals are satisfied.

Other integrity models we have investigated are distributed database integrity based on one-copy serializability and a method for surfacing memory management attributes needed to ensure that the integrity of an MLS security level is being maintained. These efforts are described in the library of models.

The Romulus availability methodology includes a model that enables a user to specify and prove availability requirements for hard real-time systems. It illustrates how Romulus can be used to specify state machines with timing information and how such timed state machines can be used to model real-time systems. In an example application of this technique, we use this kind of model to model *periodic* tasks that must be executed with a certain frequency and *sporadic* tasks that must finish execution within a certain time after they are requested.

The Romulus availability methodology also includes a model that explores some aspects of fault tolerance. It is an illustrative example of how Romulus can be used to model systems where information storage is made fault tolerant by replicating information and using a voting algorithm to determine the correct information. These examples are described in the library of models.

Chapter 2

Security

All computer systems are vulnerable to interference with the performance of the services that they provide. Maintaining the *security* of computer systems is important to ensure their proper performance. Computer security can be divided into three areas: *nondisclosure*, *integrity*, and *availability* (also known as service assurance or avoidance of denial of service). Nondisclosure deals with preventing unauthorized disclosure of information to unauthorized users. Integrity deals with protecting the information stored or transmitted in a computer system from corruption or unauthorized destruction. Availability makes sure that authorized users of a computer system can obtain access to the information and services of a computer system and do so in a timely manner.

Threats to computer systems can come from a variety of places. They can be internal due to authorized users through inadvertent actions, or to authorized users through deliberate actions, or to hardware or software failure. They can be external due to deliberate attacks by unauthorized users, or to things beyond anyone's control, such as power failures, fires, floods, etc. All computer systems should have a well defined *security policy*. This security policy should be formulated in a way that takes into account the expected threats to a system in a way appropriate for that system.

2.1 Nondisclosure

Nondisclosure is what is most often referred to by the term “computer security”. Nondisclosure concerns itself with access controls, that is, the determination of the kinds of access to information to grant to an authorized user. Nondisclosure techniques can be roughly divided into two groups. *Discretionary access control* (DAC) techniques place the control over who has what access to what information into the hands of users. Examples of DAC techniques are explicit read, write, and execute permissions granted to specific categories of users, usually the owner, the group, and everyone else, and access control lists that allow granting and denying access to a specific list of users and groups. *Mandatory access control* (MAC) techniques place the control over who has what access to what information into the hands of a system administrator or security officer. MAC techniques are most often used in the context of multilevel security where each user and each object is assigned a security level and access is granted according to the security levels of the user and the object being accessed. For example, a user should be allowed to read a file only if the user’s security level is greater than or the same as the file’s security level.

A number of models of multilevel security have been developed. The most well known of these is the Bell-LaPadula model [1]. In this model, all entities in a computer system are divided into two classifications, *subjects* and *objects*. Subjects are active entities, such as users and processes. Objects are passive containers for information, such as files. Subjects that are processes are further divided into *trusted processes* and *untrusted processes*. Users can be considered to be untrusted subjects. Every object and every subject is assigned a security level. The Bell-LaPadula model requires that two properties hold. The *simple security* property states that untrusted subjects may read only from objects of lower or equal security level, and the **-property* states that untrusted subjects may write only to objects of greater or equal security level. The Bell-LaPadula model defines rules of operations that are designed to enforce these properties. The simple security property ensures that a subject is allowed to read only information it is entitled to read. The *-property ensures that all objects are correctly labeled. The Bell-LaPadula model also introduced the concept of a *trusted subject*, a user or process exempted from the restrictions of the *-property. (For example, a process for system backup might be trusted since it needs to access information

at all security levels.) However, this model does not provide guidance for determining the security level of information leaving trusted processes, that is, that it has the correct security label.

A security model that addresses trusted processes is Goguen-Meseguer noninterference [9]. Here we will concern ourselves with one kind of trusted process, multilevel processes. These processes take in information at different security levels, process it, and output information at different security levels. We assume that each input is a discrete input labeled with a security level and that information leaves the process in the form of labeled outputs. A subject at a specific security level is restricted in its *view* of a multilevel process, that is, the subject can observe only input and output events that have equal or lower security level than the subject. All other events are *hidden* at that security level. For example, a secret subject can observe secret and unclassified events, but cannot observe top secret events. A multilevel process is secure only if low-level events do not contain any information about high-level events. Goguen-Meseguer formalized this idea for deterministic systems by requiring that the hidden high-level inputs cannot *interfere* with the sequence of low-level outputs.

Deducibility Security [38] is more general than Goguen-Meseguer noninterference and can be applied to nondeterministic systems. A system is said to be deducibility secure if any possible set of observations in the view is *consistent* with any possible sequence of hidden inputs. That is, it is impossible for a subject to “rule out” any sequence of hidden inputs. This implies that for a deducibility secure system that initially has no classified information in it, an unclassified user of that system will never learn any classified information through the system. Unfortunately, deducibility secure systems are not *composable*, that is, if one connects two deducibility secure systems together the resulting system may not be deducibility secure [27].

Romulus uses *restrictiveness* as its theory of nondisclosure security. Restrictiveness has the advantage that it is composable, that is, if two restrictive systems are combined into a single system, the resulting system will also be restrictive. There are several different definitions of restrictiveness, including trace restrictiveness [14], state restrictiveness [17, 18], and shared state restrictiveness [39]. The term *restrictiveness* generally refers to *state restrictiveness* in Romulus. Restrictiveness plays an important role in Romulus.

State restrictiveness is based on a representation of systems known as the *state machine* representation. The state machine representation models

systems that interact with their environment by exchanging events. We start with an informal definition of state machines.

A *state machine* consists of a set of *states*; a set of *events*, which are divided into three disjoint types, namely *inputs*, *outputs*, and *internal events*; a set of *initial states*; and a *state transition relation*, which is a relation that takes a state, an event, and another state and says whether it is legal for the state machine to make a transition from the first state to the second state accompanied by that event.

Two states of a state machine are equivalent if they differ only in their high-level information, that is, equivalent states are indistinguishable to low-level users. Two sequences of events are equivalent if they agree on their low-level events, that is, the subsequence consisting of the low-level inputs events is the same for each sequence. Equivalent sequences are also indistinguishable to low-level users.

Restrictiveness is a security property that requires that equivalent states be affected equivalently by equivalent input sequences, that is, they produce equivalent output sequences and end up in equivalent states. In other words, high-level inputs and high-level information cannot affect the behavior of the system as viewed by the low-level user. Restrictiveness is a generalization of Goguen-Meseguer noninterference to nondeterministic machines.

Rosenthal [35, 34] identified conditions sufficient to guarantee state restrictiveness in the broad case of buffered server processes. Brackin and Chin [4] developed a similar set of conditions known as Server-Process Restrictiveness that are equivalent to the Rosenthal conditions, but are easier to use for producing specifications and proofs. Tools for proving server-process restrictiveness of buffered server processes are included in the current Romulus release.

2.2 Integrity

Computer system integrity is a catch-all phrase to describe the assurance that data is protected from unauthorized modification or destruction. (For a general discussion of integrity see the NCSC publication “Integrity in Automated Information Systems” [19].) Our discussion of integrity is divided into two general areas. First, we consider some selected models of integrity. Second, we consider authentication protocols, listed in [19] as a technique for

establishing the identity of processes and distributing encryption keys.

2.2.1 Integrity Models

We start this section with a general discussion of integrity and then discuss integrity in Romulus.

There are a number of existing techniques to attain integrity. The most common approach to maintaining integrity is the use of discretionary access controls. These access controls restrict the modification of data to users who are explicitly given permission. The Biba model provides more structure to access control by associating degrees of integrity with users and data objects; a mandatory access control rule prevents a low-integrity user from modifying a high-integrity data object [3]. A different approach is to use type enforcement [45]. This approach associates types with each program and with each data object. The type of a program determines what type data objects can be read or modified by the program. For instance, files in a software development environment can be divided into *source files* and *object files*. The type of a compiler is given by saying it takes source files and returns object files. A word processor is allowed only to act on source files, and only object files can be executed. In this approach, integrity is associated with programs, rather than users, and the integrity is determined by functionality. One of the first attempts at a comprehensive and general policy for system integrity was the Clark-Wilson model [6]. They define a set of enforcement rules and certification rules that are designed to guarantee that only authorized users operate on trusted data and that they do so only through the use of trusted operations. The Clark-Wilson model touches on many of the elements that one would expect in a model of the informal description of integrity. However, their model does not provide a real semantics for the notions used.

The Romulus theories of integrity are a collection of formal properties that are meant to formalize various notions of *integrity* for computer systems. Romulus does not cover all possible meanings of the word “integrity”. It instead focuses on (1) formalizing some of the meanings of integrity that are currently used in the computer security community, such as the Biba model and the Clark-Wilson model, and (2) some original work on meanings of integrity that are relevant to current concerns in the distributed computing community, such as distributed data integrity and distributed authentication.

The informal meanings of integrity that we mean to capture with our

theories include the following:

- Data is trustworthy, that is, the data that users get from the system is reliable.
- Users cannot spoof the system.
- Data cannot be corrupted, that is, modified in inappropriate ways.
- Certain operations are accessible only to authorized users.

The threats to integrity we mean to capture with our theories include the following:

- Distributed information management and concurrency control problems: concurrent updates, concurrent access, interleaving of atomic parts of nonatomic transactions.
- Trojan horses and viruses.
- Modification in unprotected media (e.g., transmission media).
- Inadequate authentication.

These informal notions of integrity are formalized in Romulus in the following formal theories. These theories have not been developed to the point of application, but are included here for completeness. The first is a simple version of restrictiveness with integrity levels instead of security levels. This model stands in relation to restrictiveness as the Biba integrity model stands to the Bell-LaPadula model of security. The second is a formal theory of Clark-Wilson-like requirements that certain data be accessed only by certain operations, and that certain operations be invoked only by certain users. The third is a belief model of integrity. The fourth is a probabilistic version of deducibility security that is a first step towards being able to analyze critical systems that use encryption. These theories are described in detail in Volume II of this document set.

2.2.2 Authentication Protocols

In this section we are not concerned with general integrity. We concentrate here on a particular part of integrity that is vital to certain situations. Authentication protocols are presented in [19] as a mechanism for establishing identity and for supporting encryption. Among the aims of these protocols is the distribution of encryption keys, which is needed for cryptographic protection of data, another part of integrity.

In recent years, several logics of authentication have been developed that use *belief* logics [5, 11]. These logics enable the user to reason about the beliefs of the “principals” (i.e., the various processes) involved in a protocol. Typical beliefs are that “this key is good”, that “this message is fresh”, that “a particular principal really sent this message”, or that “a particular principal is trustworthy”. To build the logic, inference rules from the problem domain are formalized, as is the relation between protocol messages and beliefs.

Romulus incorporates an authentication logic that is based on [5, 11] and enriched by several new constructs. This logic is presented fully in Volume II of this document set. The logic has been implemented (the tool is described in the next chapter) and is used to formally verify protocol correctness.

To verify a protocol, the user lists the initial assumptions and final goals as statements in the belief logic, and describes the concrete protocol messages. Using the inference rules the user attempts to prove the protocol correct. The logic exposes the meanings of the various parts of a protocol and exposes the root of a problem if the protocol is flawed (within the scope of the logic).

2.3 Availability

Availability has broad meaning, encompassing a wide variety of issues of resource allocation and management in computer systems, real time service requirements, and fault tolerance. The literature in each of these areas is too vast to be summarized here.

The Romulus theories of availability, or service assurance, are a collection of formal properties that are meant to formalize various notions of *availability* for computer systems. Romulus does not cover all possible meanings of the term “availability”. It instead focuses on two kinds of availability: (1) requirements that services are provided in a timely manner, which implicitly

includes requirements on efficient allocation of resources in general, and (2) fault tolerance. Romulus also examines the general topic of making policies dynamic so as to facilitate tradeoffs with other requirements and reconfiguration to assure service.

The informal meanings of availability that we mean to capture with our theories include the following:

- The system will continue to function in the presence of faults.
- The system responds in a timely fashion.
- The system can reconfigure itself to optimize response.

The threats to service we mean to capture with our theories include the following:

- Hardware and software faults.
- Resource competition, both from legitimate competitors and malicious processes (e.g., “worms”).
- Unpredictable scheduling algorithms.
- Thrashing.

These informal notions of availability are formalized in Romulus in the following formal theories. The first is a general theory of fault tolerance. The second is a general approach to stating timeliness requirements. The third are dynamic versions of deducibility and restrictiveness. These theories are described in detail in Volume II of this document set.

Chapter 3

The Romulus Toolset

In this chapter, we use simple examples to introduce the Romulus tools. First, we analyze a model of a simple system using the graphical interface and the flow analyzer. Second, we formally specify and prove a piece of the model correct. Third, we describe how to specify and prove correct an authentication protocol.

3.1 The Graphical Interface

Using the Romulus graphical interface, a user designs a model and analyzes the data flow of the model. Figure 3.1 shows an example of the Romulus graphical interface. The Romulus window is divided into three areas: command buttons, a message window, and a canvas area. The command buttons are used to choose different commands, the message area displays information about the command buttons and the execution of their commands, and the canvas area is used to draw models of systems.

3.1.1 Design

For our first example, we created the design shown in Figure 3.1 using the graphical interface.

The basic objects of a Romulus design are components and ports. Components are in a tree structure; the main component being studied corresponds to a system or high-level process, and subcomponents represent subprocesses.

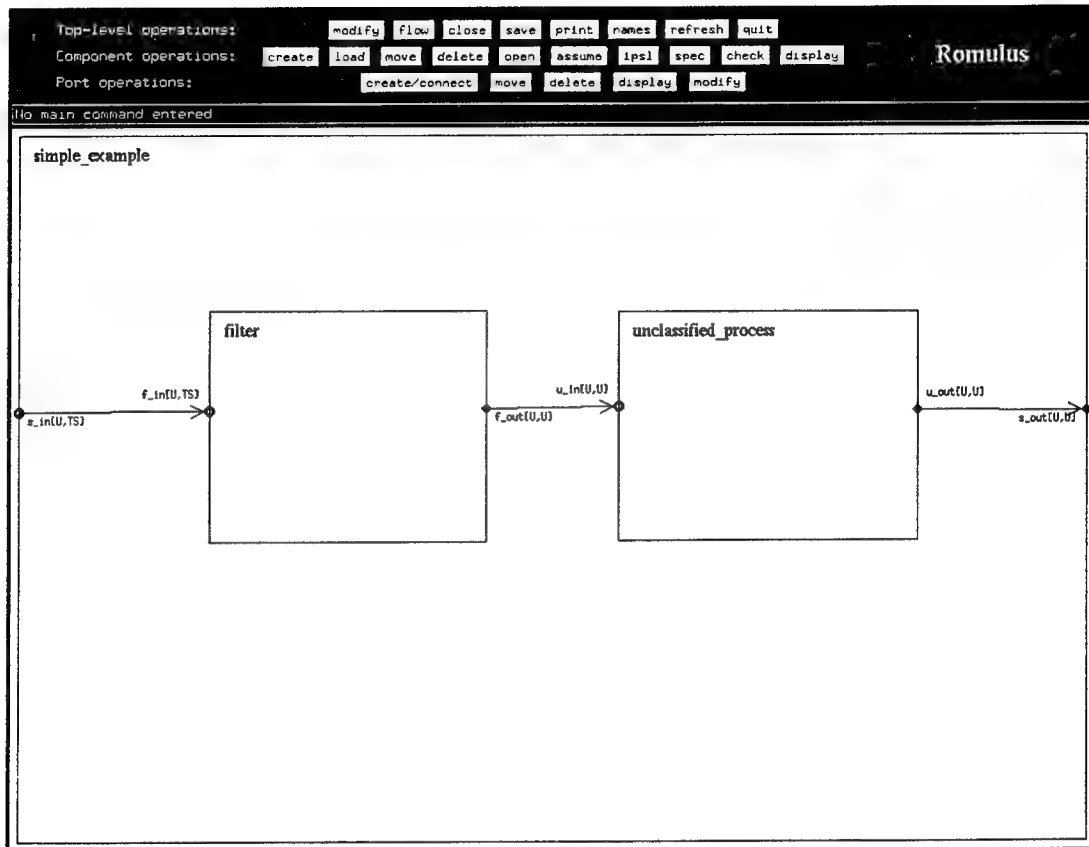


Figure 3.1: The simple example

within that system or high-level process. Ports represent data connections through which data enters or leaves the process represented by a component. A range of security levels is assigned to each port; inputs passing through an input port are assumed to have security levels in the assigned range and outputs passing through an output port are required to have security levels in the assigned range. The level range for a port is indicated in the graphics next to the port. In the example, the main component is called **simple_example** and has input port **s_in** and output port **s_out**. The subcomponents represent processes called **filter**, with one input port **f_in** and one output port **f_out**, and **unclassified_process**, with one input port **u_in** and one output port **u_out**.

We used the Component operations commands to create the subcom-

ponents. We used the **Port operations** commands to create the ports, connect ports, and assign a range of security levels to each port. We used the **Top-level operations** commands to name the components and ports and to display the port names in the canvas. We describe how to use all these commands in the Romulus User's Manual, which is Volume IV of this documentation set.

In our example, the input port labeled **f_in** receives messages from the input port labeled **s_in**. Both of these ports handle messages of all security levels in the range unclassified to top secret. This range is indicated next to these ports by the abbreviation **[U,TS]**. The **filter** component's task is to filter messages so that only unclassified messages are sent to the **unclassified_process**. The **filter**'s output port, **f_out**, has unclassified as its low and high security limits, as do the ports **u_in**, **u_out**, and **s_out**. This range is indicated next to these ports by the abbreviation **[U,U]**.

3.1.2 Flow Analysis

The Romulus flow analyzer allows the user to check a component for possible insecure data flows. A possible insecure data flow is a path from a port, through at least one component, to another port where the upper security level on the first port is higher than the lower security level on the second port. This path might be insecure because high-level information from the first port might be encoded into low-level data that reaches the second port, that is, high-level information might flow to a low-level output. To analyze the data flow of our model, we use the **Top-level operations flow** command. The flow analyzer checks the model for potential insecure data flows and, if it finds one, highlights it by drawing the affected parts using bold lines. An example of a potential insecure flow is shown in Figure 3.2.

The potential insecure flow in this case is from the port **f_in** to the port **f_out** on the **filter** process. Unless the **filter** component is proven to be secure, it might pass classified data that comes in through the input port out through the output port to the **unclassified_process**. In this simple example, only the **filter** component has a potential insecure flow.

At this point, the user can try to prove that the **filter** correctly handles data or can assume that the component is secure. If the user invokes the **Component operations assume** command and selects the **filter** component, Romulus marks the **filter** component with an asterisk in the lower right

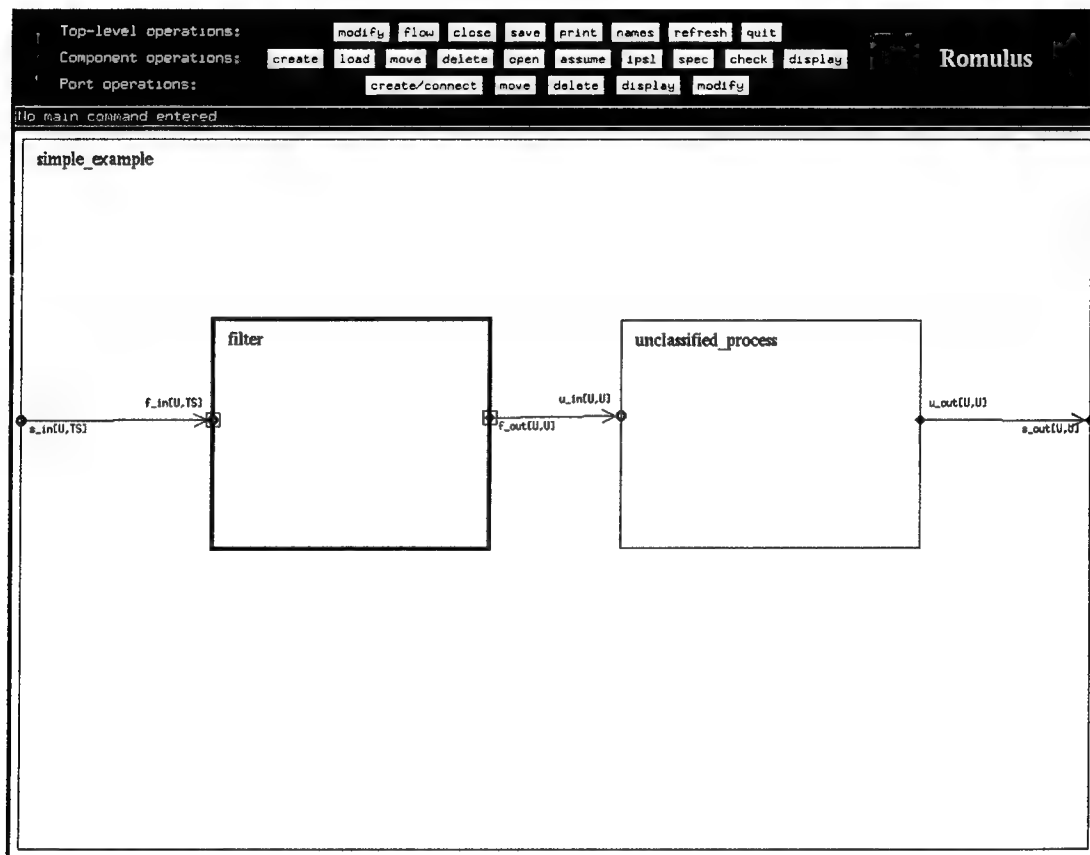


Figure 3.2: Flow analysis of the simple example

corner; this asterisk indicates that the component is assumed secure. Now when the user invokes the flow analyzer, there is no longer an insecure data flow.

In a more complex example, multiple components could have potential data flow insecurities. Romulus shows one data flow insecurity at a time. In that case, the user might consecutively assume that pieces of the model are secure to step through potential insecure flows.

Instead of assuming that a component is secure, a user has the option of trying to prove that it is. To prove that one or more components are secure, the user uses the **save** command to save the design, exits the graphical interface, and begins the steps to prove the components secure.

3.2 Proving Security

Proving that components are secure is a multi-step procedure. First, the user writes a specification for the component using the interface process specification language (IPSL). Next, the user translates the IPSL specification into the form required by the Higher Order Logic (HOL) system using the `ipsl2hol` translator, or the graphics `spec` command, and uses the HOL prover to create a HOL theory describing the process. Next, the user loads the process theory into the HOL environment and proves the desired nondisclosure property of the process. Finally, the user creates a Romulus theory, or *theory*, file that contains the information necessary for the graphical interface to verify that the process has been proved secure. If this *theory* file is loaded into the graphics, then the component for this process will be marked as having been proven secure by placing a double asterisk in the lower right corner. These steps are illustrated here; they are fully described in the Romulus User's Manual, Volume IV of this document set.

The specification for the `filter` process is given in Figure 3.3. The keywords `??Process:`, `??OutPort:`, and `??InPort:` identify the starts of process, output port, and input port specifications respectively. The `??HOL_functions:` entry gives definitions of constants that are referred to in the rest of the interface process language specification. The `??MessageVar:` entries give the names and types of the components of a message passing through the port. The `??LevelFun:` value gives the security level, as a function of the variables in an arbitrary message, of messages passing through the port. The `??LevelRange:` entry gives the range of security levels of messages passing through the port. The `??Response:` value for an input port gives the response, as a function of the variables in an arbitrary message, of the process to messages entering through that port; this response is given using the Romulus PSL formal process specification language. The `??Response:` value asserts that the filter will send a message out the port `f_out` if it is from a source whose level is `unclassified`, but otherwise ignore it, and then return to wait for the next message. The interface process specification language is fully described in Volume IV, the Romulus User's Manual.

The filter specification is translated into the form used by HOL with the command

```

??Process: filter
??HOL_functions:
new_parent "string";
new_constant
  {Name="source_level",
   Ty= ==':string->standard_level'==};

??OutPort: f_out
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: source_level source
??LevelRange: unclassified unclassified

??InPort: f_in
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: source_level source
??LevelRange: unclassified top_secret
??Response:
(If ((source_level source) = unclassified)
  (Send (f_out source data))
  Skip);;
(Call filterTop)

??EndProcess: filter

```

Figure 3.3: The filter process specification

ips12hol filter

This translation creates two files, `filter.goal.sml` and `filter.spec.sml`. The file `filter.goal.sml` is used to guide the proof process. The `filter.spec.sml` file is used to create a HOL theory file describing the filter process with the command

```
rh1 < filter.spec.sml
```

Once this specification has been completed and a HOL theory for the process has been created, the user can prove the appropriate nondisclosure property for the process.

In the case of the **filter** process, this nondisclosure property is that the process is `BNPSP_restrictive`. `BNPSP_restrictiveness` is a nondisclosure property (described in [4]) that can be applied to buffered, non-parameterized server processes.

The user starts by loading the theory file for the **filter** process, various declarations, and special purpose Romulus tactics for restrictiveness proofs. Next, the user sets the goal of proving `BNPSP_restrictiveness`.

```
g('BNPSP_restrictive
  ^filterInPred
  ^filterOutPred
  (standard_dom)
  ^filterInLevel
  ^filterOutLevel
  ^filterInvocVal
  ^filterTop');
```

The user then can do the actual proof. The Romulus tactic for proving `BNPSP_restrictiveness`, `BNPSP_restrictive.TAC`, and a standard HOL tactic, `REWRITE_TAC`, are used for this proof.

```
e(BNPSP_restrictive_TAC);
e(REWRITE_TAC [(definition "romlemmas" "standard_dom")]);
e(REWRITE_TAC [(definition "romlemmas" "standard_dom")]);
```

Finally, the user creates the `rtheory` file.

```
save_top_thm("filter_BNPSP_restrictive");
romrtheory("filter");
```

Further details can be found Volume IV.

3.3 Authentication Protocol Analysis Tool

Romulus incorporates a tool for the verification and analysis of authentication protocols. The user of this tool defines the protocol to be examined and writes a specification (requirements) of what it is intended to achieve. The user then performs a formal verification of the protocol, or in the event that the protocol is flawed, the user examines, through the logical framework, why it is flawed. Owing to the limited nature of the problem domain, this tool does not suffer from the usual problems (expense) of applying formal

methods. It is eminently practical, as the reader can see by reading the two full examples in the library of models (Volume III), where we verify and analyze standard protocols in common use. Background on authentication protocols can be found in [7, 21].

The system for verifying protocols is built within HOL, and the user works directly in HOL. We have designed a front-end language for the description and specification of protocols, and we intend that an interface for this language will be available in the next release of Romulus. This front end will enable users to define protocols using the same sort of language that is used to define protocols, in the literature, for implementors. Our front-end language is presented in Volume II. Here, we discuss the current system.

We will sketch the treatment of an example in order to describe the tool. HOL details will be omitted here for clarity. Suppose the user wishes to verify the familiar Denning-Sacco key distribution protocol [7]. This protocol is presented in the literature like so:

1. $A \rightarrow S: A, B;$
2. $S \rightarrow A: \{B, k, t, \{A, k, t\}_{e(kb)}}_{e(ka)};$
3. $A \rightarrow B: \{A, k, t\}_{e(kb)};$

The notation $A \rightarrow B: M$ means that principal A sends principal B the message M. Here messages are preceded with sequence numbers for reference purposes.

In this protocol there are three messages sent between principals A, B, and S. S is a key server, and A and B are clients who wish to obtain a secret key that they can use for encryption of messages sent between them. For clients A and B, k is the session key chosen by the server S; The timestamp t is taken from the server's clock. A and S share the secret key ka , and B and S share the secret key kb . Finally, $\{\dots\}_{e(kb)}$ is the result of encrypting the contents of the curly braces with the key kb .

3.3.1 Describing and Specifying a Protocol

Definitions of some of the objects necessary for describing and specifying protocols are contained in a HOL theory `crypto_90`. These objects are the belief predicate, the types of the principals and messages, and the encryption function. Various axioms in this theory determine needed relationships between the objects. Of special significance is the turnstile of our logic —

the predicate `theorem`, which indicates that a HOL term is a theorem in our logic.

Other objects necessary for describing and specifying a protocol are specific to the protocol. The user defines these objects in another HOL theory. The user defines the protocol in this theory using the protocol specification language. For example,

```
send A S ((name A) APP (name B))
send S A (encrypt ka ((name B) APP k APP t APP
                     (encrypt kb ((name A) APP k APP t))))
send A B (encrypt kb ((name A) APP k APP t))
```

Here `APP` is the means of appending separate message components and `(name A)` is the name of principal A.

The user must also define the initial assumptions needed for the protocol; for this protocol, some of these assumptions are

```
theorem(believes A (is_shared_secret A S ka))
theorem(believes A (is_fresh t))
```

The first assumption states A's belief that the key `ka` is known only to A and S, and that no one else could guess it. The second states A's belief that the timestamp A has used is indeed recently generated.

Finally, the user states what the user hopes to achieve by executing the protocol. This statement, called `postcondition`, is a conjunction of a number of things like:

```
theorem(believes A (is_shared_secret A B k))
```

together with the claim that the key `k` is freshly generated, etc.

3.3.2 Proving a Protocol

The user wants this `postcondition` to be a theorem of the authentication logic and declares this `postcondition` to be goal of a HOL proof session. A proof is then carried out, using the inference rules of our logic, which are contained in the theory `crypto_90`.

To get the flavor of the inference rules, we will look at one:

```
theorem(believes p (is_fresh x)) /\ theorem(possesses p k)
==>
theorem(believes p (is_fresh (encrypt k x)))
```

This rule reflects one thing we can deduce, based partly on the properties of encryption functions. That is, if principal p believes that the text x is recently generated and p possesses the encryption key k , then p believes that x encrypted with key k is also recently generated. This rule is used to show that a message containing an encrypted version of a fresh string is itself fresh.

If the proof succeeds, the protocol meets its specification. It can happen that the proof cannot be completed because the protocol is flawed. In this case, it becomes clear from the logic where the problem is.

The theory and use of the tool are described fully in Volume II and Volume IV respectively. Detailed examples of the analysis of two protocols are presented in the library of models, Volume III. The design for the front-end language is described in Volume II.

Chapter 4

Library of Models

The Romulus library of models is a collection of examples that provide insight into building and analyzing models for security properties. The library includes examples from the areas of nondisclosure, integrity, and availability. These properties are described in Volume II of the Romulus documentation set. Because the areas of nondisclosure, integrity, and availability are so broad, our models address specific aspects of each area. For nondisclosure, the focus is on mandatory access control, and in particular, the restrictiveness theory. For integrity, the models handle label integrity (network driver), correctness of distributed authentication, and a method of achieving one-copy serializability (distributed databases). For availability, the models address fault tolerance and real-time scheduling.

We give a brief description of each model below. Complete descriptions of each model can be found in Volume III of the Romulus documentation set.

4.1 Abstract Guard

The abstract guard is a generic description of a process that filters out messages that should not be delivered to processes operating in a given security range. This model models the use of a shared resource (shared directories); we use a variation on restrictiveness, called shared-state restrictiveness, to show that it is secure with respect to nondisclosure.

The generic guard process formalized in Volume III is very abstract, modeling virtually any program for analyzing messages and choosing which mes-

sages can be released. In this model, input messages are placed into an input directory. The guard examines the security level of each input message and, if it is safe, (i.e., if it can be released to processes in the specified security range) puts it into an output directory. If it is not safe, the guard places the message into an audit directory. The guard design guarantees that only one message is accessed at a time and that if a new (or moved) message is created from an old one the new message has the same or higher level as the old one.

We prove that a simple security property is sufficient to guarantee shared-state restrictiveness. Although this property is rather strong, we believe it could be shown to follow from simpler and more specific properties of guard components — properties that would be easier to establish in themselves than shared-state or regular restrictiveness. These properties could then be used as guides by the designers of actual guards.

4.2 MINIX

The MINIX model is a specification for a secure (i.e., restrictive) operating system based on the MINIX operating system [40]. The main additions to MINIX in the system modeled here are data structures for maintaining security-level information and procedures for imposing mandatory access controls. The system model given here is believed to be restrictive, but we do not give a full proof of restrictiveness.

The security model used in the operating system modeled here is for the most part standard. The active “subjects” in the model are user processes, and the “objects” are data files or user processes. (The `kill` command provides an example of treating a user process as an object.) Subjects and objects are assigned security levels, and a subject is not permitted to obtain information about an object unless the subject’s level dominates the object’s level or permitted to affect an object unless the object’s level dominates the subject’s level. Since the model assumes that security levels are given by information stored in the operating system itself, the model might be generalized to describe an operating system that allows security levels to be changed dynamically. Also, since the model’s security is analyzed in terms of restrictiveness in the special case of buffered server processes, the model also implicitly addresses possible “covert channels” involving shared resources

such as system data tables.

The operating system model given here is primarily of interest because when the proof that it is restrictive is completed, it will provide insight into the design of restrictive operating systems. If proven restrictive, it will show that even something as complicated as this model of an operating system can be made restrictive. The model and the proofs of its properties given here also serve as nice examples of how the Romulus techniques and utilities for specifying processes and proving facts about them can be used to manage complexity and simplify proofs.

4.3 Network Driver

A secure network device driver [32] must ensure that the integrity of MLS information is maintained across the network. In the network driver described in [32], a security level is added to a packet before it goes out to the network, and this information must be used to route the packet to the appropriate security level when it arrives at its destination. Maintaining the integrity of the label is essential in order for this scheme to maintain security. In particular, care must be taken to ensure that the label on a packet is not corrupted between the time that the packet is labeled and the time that it is sent out over the network. Ensuring that the label is not corrupted requires careful memory management techniques.

In the secure network driver, a labeled packet is stored in a buffer while it is waiting to be sent out over the network. If a process other than the sending process, the labeler, or network driver is allowed access to this buffer, then the label could be corrupted. Corruption could occur in a number of different ways. For example, more than one process could be allocated the same piece of memory, allowing one of the processes to inadvertently overwrite the label in a buffer belonging to another process. Or, a process could maintain a pointer to a piece of memory that it had deallocated and that had been allocated to another process, again allowing that process to inadvertently overwrite the label in a buffer belonging to the other process. Or, a process might purposely seek out the memory allocated to other processes in hopes of deliberately changing the labels on high-level packets so that a low-level confederate on another node of the network could gain access to the packets.

The network driver model describes a method for managing these memory

management issues to ensure that the integrity of an MLS security level is being maintained. This integrity property is used in the security argument that the network labeler and network driver correctly set and maintain the security level. Included in this argument are formal specifications of the relevant parts of the network labeler and its properties.

4.4 Authentication Protocols

We present an analysis of two authentication protocols, the Denning-Sacco protocol and the Needham-Schroeder protocol. Authentication protocols are important contributors to integrity assurance because they are used to establish the correct identity of processes and distribute encryption keys. An authentication protocol is an exchange of messages between a number of processes, called “principals”. A typical aim of a protocol is to establish an encryption key shared by two principals. The message exchange often involves a third party — a “keyserver”— who is trusted to generate good keys and keep secrets. Protocol messages employ a variety of techniques to ensure the identity of a principal, that the messages have been recently generated, and that the keys exchanged are protected.

We deal here with protocols modeled at the level of the messages between the processes (or “principals”) involved, as is standard in this area. We use the Romulus implementation of authentication logic to state requirements on particular protocols and prove that they are correct, or in the case of an inadequate protocol, examine what the protocol lacks and what it *can* establish.

For each protocol, we present the description of the protocol and its specification, and we describe the proof of correctness. We perform the analysis in the Romulus implementation of authentication logic. This logic is a belief logic, which enables us to express the belief-states of principals and prove that after protocol execution, the principals are operating with a correct set of beliefs about the identity of other principals and the adequacy of encryption keys.

4.5 Distributed Database

The distributed database model describes a method of building multilevel databases that takes into account both integrity and nondisclosure requirements, together with a formal specification of the trusted part of the method. The trusted part of the method is a protocol developed by Kogan and Jajodia [12] for distributed, replicated databases; this protocol is both restrictive and satisfies the integrity property, one-copy serializability [2].

Serializability is the standard integrity property for concurrency control in databases. Serializability ensures that if a number of transactions are interleaved, then the effect will be equivalent to some scenario in which the transactions are not interleaved. When dealing with distributed databases, we need a modified form of serializability, called one-copy serializability, to handle the fact that there are multiple versions of the same information on different hosts.

The replicated database described in [12] consists of a number of different interconnected databases called *containers*. Each individual database contains all of the information at or below some security level, that is, data in lower level containers is replicated in higher level containers. Each container is single-level, and information is not permitted to flow from a high-level database to a low-level database. Any number of containers can reside on a single host. This database model uses an MLS approach, where all of the synchronization responsibility between databases on a particular host is placed into a single trusted process. This approach may be able to more efficiently manage shared updates.

For integrity, control is needed on the propagation of information from low to high levels to preserve database consistency. This control must be done, in order to achieve nondisclosure, in a way such that there is no communication from higher level containers to lower level containers. The protocol described in [12] has the desired integrity and nondisclosure properties. We provide a formal specification of the trusted part of their protocol.

4.6 Fault Tolerant Reference Monitor

The Fault Tolerant Reference Monitor (FTRM) is designed to support multiple clients with arbitrary security classifications accessing data in a hierarchi-

cal file system with files and directories at arbitrary security classifications. It is intended to implement an access control policy that allows clients to obtain information only at an equal or lower security level, or output information only at an equal or greater security level. The FTRM accomplishes fault tolerant multilevel security by replicating files and tables of security levels on multiple network nodes. The replicated data is used to mask data corruption faults by implementing a collection of voting algorithms. The FTRM is designed so that any number of faults can be withstood if there are sufficiently many nodes and sufficient replication of file system data and security level tables. In particular, secure mediation of access to data will be unaffected by faults if there is a sufficient degree of replication.

The FTRM model is designed to satisfy both nondisclosure and availability security properties. Nondisclosure security is assured by designing the FTRM to be restrictive. Availability security is assured by designing the FTRM to be fault tolerant with respect to various kinds of faults in the nodes on the network, including nodes crashing, disks and other devices for maintaining the file system crashing, and corruption of data (either in ordinary files or in files containing records of client and file security levels) on disks.

Faults are modeled as inputs to the system; by assigning the security level systemhigh to these inputs, the fault tolerance properties of the FTRM can be established by showing it to be restrictive. This model includes a complete specification of the FTRM, but a proof of restrictiveness has not been done.

4.7 Real-Time Scheduler

The purpose of this model is to explore some aspects of availability requirements for hard real-time systems. It is an illustrative example of how *Romulus* can be used to specify state machines with timing information, and how such timed state machines can be used to model real-time systems. In this example, the real-time system is required to schedule two representative tasks similar to those handled by the Operational Flight Program (OFP) of the A-7E Navy aircraft:

- a *periodic* task, which repeatedly updates a navigational database with the current position of the aircraft, and

- a *sporadic* task, which is initiated at the request of the pilot and fires a missile.

These tasks must be executed in such a way that the availability requirements of both are met.

A hard real-time system is one designed to meet requirements not only on what actions it performs, but also when it performs them. Such a system must schedule processes to perform tasks that are *time-critical* (i.e., they must be performed in a “timely” fashion).

In the context of a hard real-time system, availability properties relate to the timeliness of time-critical tasks and are expressed as constraints on the timing of processes. Timing constraints can of course be arbitrarily complex. However, in this notoriously difficult (and hazardous) field, there have evolved certain useful models of processing and their hard real-time requirements. These models are both general and powerful enough to deal with a wide variety of real-time processing needs, in particular for avionics processing. They are also simple enough to admit full analysis and safe implementation.

The real-time scheduler model models the scheduling of one periodic task and one sporadic task using a static priority interrupt scheduling algorithm [41]. The sporadic task is given a higher priority than the periodic task. The requirements on this model are that each task completes execution within a fixed period of time after it is requested. The periodic task is requested at fixed intervals; the sporadic task can be requested at any time, but two requests must be separated by a minimum fixed interval. This model includes a proof that the sporadic task meets its requirements and an almost completed proof that the periodic task meets its requirements.

Bibliography

- [1] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, Revision 2, MITRE Corp., Bedford MA, March 1976.
- [2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [3] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford MA, April 1977.
- [4] S. Brackin and S-K Chin. Server-process restrictiveness in HOL. In *HOL User's Group Workshop*, Vancouver, Canada, August 1993. Springer Verlag.
- [5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1), February 1990.
- [6] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1987.
- [7] D.E.Denning and G.M. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533–536, August 1981.
- [8] Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD-5200.28-STD.
- [9] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE.

- [10] Li Gong and Geoffrey Hird. The ORA toolkit for analyzing cryptographic protocols: A user manual. (Draft), 1993.
- [11] Li Gong, Roger Needham, and Raphael Yahalom. A methodology for analyzing cryptographic protocols. In *Proceedings of 11th IEEE Symposium on Research in Security and Privacy*, pages 234–248, Oakland, CA, May 1990.
- [12] Boris Kogan and Sushil Jajodia. Data replication and multilevel-secure transaction processing, June 1993. Manuscript.
- [13] Tatiana Korelsky et al. Ulysses: A computer security modeling environment. In *Proceedings of the 11th National Computer Security Conference*, pages 20–28, Baltimore, MD, October 1988.
- [14] Daryl McCullough. Specifications for multilevel security and a hook-up property. In *Proceedings of the Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987. IEEE.
- [15] Daryl McCullough. Foundations of Ulysses: The theory of security. Technical Report RADC-TR-87-222, Rome Air Development Center, May 1988.
- [16] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE.
- [17] Daryl McCullough. Security analysis of a token ring using Ulysses. In *Proceedings of the Fourth Annual Conference on Computer Assurance (COMPASS '89)*, pages 113–118, Gaithersburg, MD, June 1989.
- [18] Daryl McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.
- [19] National Computer Security Center. *Integrity in Automated Information Systems*, September 1991.
- [20] National Computer Security Center. *A Guide to Understanding Security Modeling in Trusted Systems*, October 1992. NCSC-TG-010, Version-1.

- [21] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, December 1978.
- [22] ORA. Romulus: A computer security properties modeling environment, final report - volume 1: Overview. Technical report, ORA, June 1990.
- [23] ORA. Romulus: A computer security properties modeling environment, final report - volume 10: Knowledge base component. Technical report, ORA, June 1990.
- [24] ORA. Romulus: A computer security properties modeling environment, final report - volume 2a and 2b: Theory of security and mathesis. Technical report, ORA, June 1990.
- [25] ORA. Romulus: A computer security properties modeling environment, final report - volume 3: Specification languages. Technical report, ORA, June 1990.
- [26] ORA. Romulus: A computer security properties modeling environment, final report - volume 4: Security modeling support. Technical report, ORA, June 1990.
- [27] ORA. Romulus: A computer security properties modeling environment, final report - volume 5: Mathematical component. Technical report, ORA, June 1990.
- [28] ORA. Romulus: A computer security properties modeling environment, final report - volume 6: Natural language generator. Technical report, ORA, June 1990.
- [29] ORA. Romulus: A computer security properties modeling environment, final report - volume 7: Link and integration. Technical report, ORA, June 1990.
- [30] ORA. Romulus: A computer security properties modeling environment, final report - volume 8: Library of models. Technical report, ORA, June 1990.
- [31] ORA. Romulus: A computer security properties modeling environment, final report - volume 9: Link to an existing formal specification language. Technical report, ORA, June 1990.

- [32] ORA. A secure network device driver, final report. Technical report, ORA, November 1990.
- [33] ORA. Romulus course notes, 1992.
- [34] David Rosenthal. An approach to increasing the automation of the verification of security. In *Proceedings of Computer Security Foundations Workshop*, pages 90–97, Franconia, NH, June 1988. The MITRE Corporation, M88-37.
- [35] David Rosenthal. Implementing a verification methodology for McCullough security. In *Proceedings of Computer Security Foundations Workshop II*, pages 133–140, Franconia, NH, June 1989. IEEE.
- [36] David Rosenthal. Security models for priority buffering and interrupt handling. In *Proceedings of Computer Security Foundations Workshop III*, pages 91–97, Franconia NH, June 1990. IEEE Computer Society Press.
- [37] Konrad Slind. *Hol90 Users Manual, Preliminary Version*.
- [38] Ian Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183, September 1986.
- [39] Ian Sutherland. Shared-state restrictiveness. ORA Internal Report, July 1992.
- [40] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
- [41] Thomas J. Teixeira. Static priority interrupt scheduling. In *Proceedings of the 7th Texas Conference on Computing Systems*. University of Houston, 1978.
- [42] University of Cambridge, DSTO, and SRI International. *The HOL System Description*.
- [43] University of Cambridge, DSTO, and SRI International. *The HOL System Reference*.

- [44] University of Cambridge, DSTO, and SRI International. *The HOL System Tutorial*.
- [45] Raymond M. Wong and Y. Eugene Ding. Providing software integrity using type managers. In *Proceedings of Fourth Aerospace Computer Security Applications Conference*, pages 287–294, Orlando, FL, December 1988. IEEE Computer Society Press.

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.